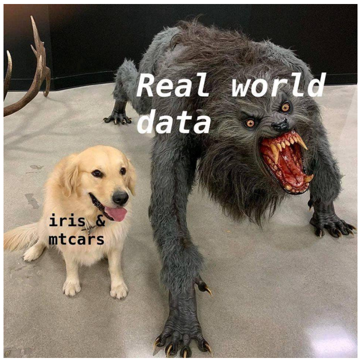
7 Data Wrangling



7.1 Learning Objectives

Basic

1. Be able to use the 6 main dplyr one-table verbs: ▶
    - `select()`
    - `filter()`
    - `arrange()`
    - `mutate()`
    - `summarise()`
    - `group_by()`
2. Be able to wrangle data by chaining tidyr and dplyr functions ▶
3. Be able to use these additional one-table verbs: ▶
    - `rename()`
    - `distinct()`
    - `count()`
    - `slice()`
    - `pull()`

Intermediate

4. Fine control of `select()` operations ▶
5. Use window functions ▶

7.2 Setup

```
# libraries needed for these examples
library(tidyverse)
library(lubridate)
library(reprores)
set.seed(8675309) # makes sure random numbers are reproducible
```

7.2.1 The `disgust` dataset

These examples will use data from `reprores::disgust`, which contains data from the Three Domain Disgust Scale. Each participant is identified by a unique `user_id` and each questionnaire completion has a unique `id`. Look at the Help for this dataset to see the individual questions.

```
data("disgust", package = "reprores")

#disgust <-
read_csv("https://psyteachr.github.io/reprores/data/disgust.csv")
```

7.3 Six main dplyr verbs

Most of the data wrangling you'll want to do with psychological data will involve the `tidyr` functions you learned in Chapter 4 and the six main `dplyr` verbs: `select`, `filter`, `arrange`, `mutate`, `summarise`, and `group_by`.

7.3.1 select()

Select columns by name or number.

You can select each column individually, separated by commas (e.g., `col1, col2`). You can also select all columns between two columns by separating them with a colon (e.g., `start_col:end_col`).

```
moral <- disgust %>% select(user_id, moral1:moral7)
names(moral)
```

```
## [1] "user_id" "moral1"  "moral2"  "moral3"  "moral4"  "moral5"
"moral6"
## [8] "moral7"
```

You can select columns by number, which is useful when the column names are long or complicated.

```
sexual <- disgust %>% select(2, 11:17)
names(sexual)
```

```
## [1] "user_id" "sexual1" "sexual2" "sexual3" "sexual4" "sexual5"
"sexual6"
## [8] "sexual7"
```

You can use a minus symbol to unselect columns, leaving all of the other columns. If you want to exclude a span of columns, put parentheses around the span first (e.g., `-(moral1:moral7)`, not `-moral1:moral7`).

```
pathogen <- disgust %>% select(-id, -date, -(moral1:sexual7))
names(pathogen)
```

```
## [1] "user_id"   "pathogen1" "pathogen2" "pathogen3" "pathogen4"
"pathogen5"
## [7] "pathogen6" "pathogen7"
```

7.3.1.1 Select helpers

You can select columns based on criteria about the column names.

7.3.1.1.1 `starts_with()`

Select columns that start with a character string.

```
u <- disgust %>% select(starts_with("u"))
names(u)
```

```
## [1] "user_id"
```

7.3.1.1.2 `ends_with()`

Select columns that end with a character string.

```
firstq <- disgust %>% select(ends_with("1"))
names(firstq)
```

```
## [1] "moral1"    "sexual1"   "pathogen1"
```

7.3.1.1.3 `contains()`

Select columns that contain a character string.

```
pathogen <- disgust %>% select(contains("pathogen"))
names(pathogen)
```

```
## [1] "pathogen1" "pathogen2" "pathogen3" "pathogen4" "pathogen5"
"pathogen6"
## [7] "pathogen7"
```

7.3.1.1.4 `num_range()`

Select columns with a name that matches the pattern `prefix`.

```
moral2_4 <- disgust %>% select(num_range("moral", 2:4))
names(moral2_4)
```

```
## [1] "moral2" "moral3" "moral4"
```

Use `width` to set the number of digits with leading zeros. For example, `num_range('var_', 8:10, width=2)` selects columns `var_08`, `var_09`, and `var_10`.

7.3.2 filter()

Select rows by matching column criteria.

Select all rows where the user_id is 1 (that's Lisa).

```
disgust %>% filter(user_id == 1)
```

id	user_id	date	moral1	moral2	moral3	moral4	moral5	moral6	moral7	sexu
1	1	2008-07-10	2	2	1	2	1	1	1	

Remember to use `==` and not `=` to check if two things are equivalent. A single `=` assigns the righthand value to the lefthand variable and (usually) evaluates to `TRUE`.

You can select on multiple criteria by separating them with commas.

```
amoral <- disgust %>% filter(
  moral1 == 0,
  moral2 == 0,
  moral3 == 0,
  moral4 == 0,
  moral5 == 0,
  moral6 == 0,
  moral7 == 0
)
```

You can use the symbols `&`, `|`, and `!` to mean "and", "or", and "not". You can also use other operators to make equations.

```
# everyone who chose either 0 or 7 for question moral1
moral_extremes <- disgust %>%
  filter(moral1 == 0 | moral1 == 7)

# everyone who chose the same answer for all moral questions
moral_consistent <- disgust %>%
  filter(
    moral2 == moral1 &
    moral3 == moral1 &
    moral4 == moral1 &
    moral5 == moral1 &
    moral6 == moral1 &
    moral7 == moral1
  )

# everyone who did not answer 7 for all 7 moral questions
moral_no_ceiling <- disgust %>%
  filter(moral1+moral2+moral3+moral4+moral5+moral6+moral7 != 7*7)
```

7.3.2.1 Match operator (%in%)

Sometimes you need to exclude some participant IDs for reasons that can't be described in code. The match operator (`%in%`) is useful here for testing if a column value is in a list. Surround the equation with parentheses and put `!` in front to test that a value is not in the list.

```
no_researchers <- disgust %>%
  filter(!(user_id %in% c(1,2)))
```

7.3.2.2 Dates

You can use the `lubridate` package to work with dates. For example, you can use the `year()` function to return just the year from the `date` column and then select only data collected in 2010.

```
disgust2010 <- disgust %>%
  filter(year(date) == 2010)
```

id	user_id	date	moral1	moral2	moral3	moral4	moral5	moral6	moral7	se
6902	5469	2010-12-06	0	1	3	4	1	0	1	
6158	6066	2010-04-	4	5	6	5	5	4	4	

		18							
6362	7129	2010-06-09	4	4	4	4	3	3	2
6302	39318	2010-05-20	2	4	1	4	5	6	0
5429	43029	2010-01-02	1	1	1	3	6	4	2
6732	71955	2010-10-15	2	5	3	6	3	2	5

Table 7.1: Rows 1-6 from `disgust2010`

Or select data from at least 5 years ago. You can use the `range` function to check the minimum and maximum dates in the resulting dataset.

```
disgust_5ago <- disgust %>%
  filter(date < today() - dyears(5))

range(disgust_5ago$date)
```

```
## [1] "2008-07-10" "2017-04-04"
```

7.3.3 arrange()

Sort your dataset using `arrange()`. You will find yourself needing to sort data in R much less than you do in Excel, since you don't need to have rows next to each other in order to, for example, calculate group means. But `arrange()` can be useful when preparing data from display in tables.

```
disgust_order <- disgust %>%
  arrange(date, moral1)
```

id	user_id	date	moral1	moral2	moral3	moral4	moral5	moral6	moral7	sexu
1	1	2008-07-10	2	2	1	2	1	1	1	
3	155324	2008-07-11	2	4	3	5	2	1	4	
6	155386	2008-	2	4	0	4	0	0	0	

		07-12							
7	155409	2008-07-12	4	5	5	4	5	1	5
4	155366	2008-07-12	6	6	6	3	6	6	6
5	155370	2008-07-12	6	6	4	6	6	6	6

Table 7.2: Rows 1-6 from `disgust_order`

Reverse the order using `desc()`

```
disgust_order_desc <- disgust %>%
  arrange(desc(date))
```

id	user_id	date	moral1	moral2	moral3	moral4	moral5	moral6	moral7
39456	356866	2017-08-21	1	1	1	1	1	1	1
39447	128727	2017-08-13	2	4	1	2	2	5	3
39371	152955	2017-06-13	6	6	3	6	6	6	6
39342	48303	2017-05-22	4	5	4	4	6	4	5
39159	151633	2017-04-04	4	5	6	5	3	6	2
38942	370464	2017-02-01	1	5	0	6	5	5	5

Table 7.3: Rows 1-6 from `disgust_order_desc`

7.3.4 mutate()

Add new columns. This is one of the most useful functions in the tidyverse.

Refer to other columns by their names (unquoted). You can add more than one column in the same mutate function, just separate the columns with a comma. Once you make a new column, you can use it in further column definitions e.g., `total` below).

```
disgust_total <- disgust %>%
  mutate(
    pathogen = pathogen1 + pathogen2 + pathogen3 + pathogen4 +
pathogen5 + pathogen6 + pathogen7,
    moral = moral1 + moral2 + moral3 + moral4 + moral5 + moral6 +
moral7,
    sexual = sexual1 + sexual2 + sexual3 + sexual4 + sexual5 + sexual6
+ sexual7,
    total = pathogen + moral + sexual,
    user_id = paste0("U", user_id)
  )
```

id	user_id	date	moral1	moral2	moral3	moral4	moral5	moral6	moral7
1199	U0	2008-10-07	5	6	4	6	5	5	6
1	U1	2008-07-10	2	2	1	2	1	1	1
1599	U2	2008-10-27	1	1	1	1	NA	NA	1
13332	U2118	2012-01-02	0	1	1	1	1	2	1
23	U2311	2008-07-15	4	4	4	4	4	4	4
1160	U3630	2008-10-06	1	5	NA	5	5	5	1

Table 7.4: Rows 1-6 from `disgust_total`

> You can overwrite a column by giving a new column the same name as the old column (see `user_id`) above. Make sure that you mean to do this and that you aren't trying to use the old column value after you redefine it.

7.3.5 summarise()

Create summary statistics for the dataset. Check the Data Wrangling Cheat Sheet or the Data Transformation Cheat Sheet for various summary functions. Some common ones are: `mean()`, `sd()`, `n()`, `sum()`, and `quantile()`.

```
disgust_summary<- disgust_total %>%
  summarise(
    n = n(),
    q25 = quantile(total, .25, na.rm = TRUE),
    q50 = quantile(total, .50, na.rm = TRUE),
    q75 = quantile(total, .75, na.rm = TRUE),
    avg_total = mean(total, na.rm = TRUE),
    sd_total  = sd(total, na.rm = TRUE),
    min_total = min(total, na.rm = TRUE),
    max_total = max(total, na.rm = TRUE)
  )
```

n	q25	q50	q75	avg_total	sd_total	min_total	max_total
20000	59	71	83	70.6868	18.24253	0	126

Table 7.5: All rows from `disgust_summary`

7.3.6 group_by()

Create subsets of the data. You can use this to create summaries, like the mean value for all of your experimental groups.

Here, we'll use `mutate` to create a new column called `year`, group by `year`, and calculate the average scores.

```
disgust_groups <- disgust_total %>%
  mutate(year = year(date)) %>%
  group_by(year) %>%
  summarise(
    n = n(),
    avg_total = mean(total, na.rm = TRUE),
    sd_total  = sd(total, na.rm = TRUE),
    min_total = min(total, na.rm = TRUE),
    max_total = max(total, na.rm = TRUE),
    .groups = "drop"
  )
```

year	n	avg_total	sd_total	min_total	max_total

2008	2578	70.29975	18.46251	0	126
2009	2580	69.74481	18.61959	3	126
2010	1514	70.59238	18.86846	6	126
2011	6046	71.34425	17.79446	0	126
2012	5938	70.42530	18.35782	0	126
2013	1251	71.59574	17.61375	0	126
2014	58	70.46296	17.23502	19	113
2015	21	74.26316	16.89787	43	107
2016	8	67.87500	32.62531	0	110
2017	6	57.16667	27.93862	21	90

Table 7.6: All rows from `disgust_groups`

> If you don't add `.groups = "drop"` at the end of the `summarise()` function, you will get the following message: "`summarise()` ungrouping output (override with `.groups` argument)". This just reminds you that the groups are still in effect and any further functions will also be grouped.
>
> Older versions of dplyr didn't do this, so older code will generate this warning if you run it with newer version of dplyr. Older code might `ungroup()` after `summarise()` to indicate that groupings should be dropped. The default behaviour is usually correct, so you don't need to worry, but it's best to explicitly set `.groups` in a `summarise()` function after `group_by()` if you want to "keep" or "drop" the groupings.

You can use `filter` after `group_by`. The following example returns the lowest total score from each year (i.e., the row where the `rank()` of the value in the column `total` is equivalent to `1`).

```
disgust_lowest <- disgust_total %>%
  mutate(year = year(date)) %>%
  select(user_id, year, total) %>%
  group_by(year) %>%
  filter(rank(total) == 1) %>%
  arrange(year)
```

user_id	year	total
U236585	2009	3

U292359	2010	6
U245384	2013	0
U206293	2014	19
U407089	2015	43
U453237	2016	0
U356866	2017	21

Table 7.7: All rows from `disgust_lowest`

You can also use `mutate` after `group_by`. The following example calculates subject-mean-centered scores by grouping the scores by `user_id` and then subtracting the group-specific mean from each score. Note the use of `gather` to tidy the data into a long format first.

```
disgust_smc <- disgust %>%
  gather("question", "score", moral1:pathogen7) %>%
  group_by(user_id) %>%
  mutate(score_smc = score - mean(score, na.rm = TRUE)) %>%
  ungroup()
```

> Use `ungroup()` as soon as you are done with grouped functions, otherwise the data table will still be grouped when you use it in the future.

id	user_id	date	question	score	score_smc
1199	0	2008-10-07	moral1	5	0.9523810
1	1	2008-07-10	moral1	2	0.0476190
1599	2	2008-10-27	moral1	1	0.0000000
13332	2118	2012-01-02	moral1	0	-3.0000000
23	2311	2008-07-15	moral1	4	0.6190476
1160	3630	2008-10-06	moral1	1	-1.2500000

Table 7.8: Rows 1-6 from `disgust_smc`

7.3.7 All Together

A lot of what we did above would be easier if the data were tidy, so let's do that first. Then we can use `group_by` to calculate the domain scores.

After that, we can spread out the 3 domains, calculate the total score, remove any rows with a missing (`NA`) total, and calculate mean values by year.

```
disgust_tidy <- reprores::disgust %>%
  gather("question", "score", moral1:pathogen7) %>%
  separate(question, c("domain","q_num"), sep = -1) %>%
  group_by(id, user_id, date, domain) %>%
  summarise(score = mean(score), .groups = "drop")
```

id	user_id	date	domain	score
1	1	2008-07-10	moral	1.428571
1	1	2008-07-10	pathogen	2.714286
1	1	2008-07-10	sexual	1.714286
3	155324	2008-07-11	moral	3.000000
3	155324	2008-07-11	pathogen	2.571429
3	155324	2008-07-11	sexual	1.857143

Table 7.9: Rows 1-6 from `disgust_tidy`

```
disgust_scored <- disgust_tidy %>%
  spread(domain, score) %>%
  mutate(
    total = moral + sexual + pathogen,
    year = year(date)
  ) %>%
  filter(!is.na(total)) %>%
  arrange(user_id)
```

id	user_id	date	moral	pathogen	sexual	total	year
1199	0	2008-10-07	5.285714	4.714286	2.142857	12.142857	2008
1	1	2008-07-10	1.428571	2.714286	1.714286	5.857143	2008
13332	2118	2012-01-02	1.000000	5.000000	3.000000	9.000000	2012
23	2311	2008-07-15	4.000000	4.285714	1.857143	10.142857	2008
7980	4458	2011-09-05	3.428571	3.571429	3.000000	10.000000	2011
552	4651	2008-08-23	3.857143	4.857143	4.285714	13.000000	2008

Table 7.10: Rows 1-6 from `disgust_scored`

```
disgust_summarised <- disgust_scored %>%
  group_by(year) %>%
  summarise(
    n = n(),
    avg_pathogen = mean(pathogen),
    avg_moral = mean(moral),
    avg_sexual = mean(sexual),
    first_user = first(user_id),
    last_user = last(user_id),
    .groups = "drop"
  )
```

year	n	avg_pathogen	avg_moral	avg_sexual	first_user	last_user
2008	2392	3.697265	3.806259	2.539298	0	188708
2009	2410	3.674333	3.760937	2.528275	6093	251959
2010	1418	3.731412	3.843139	2.510075	5469	319641
2011	5586	3.756918	3.806506	2.628612	4458	406569
2012	5375	3.740465	3.774591	2.545701	2118	458194
2013	1222	3.771920	3.906944	2.549100	7646	462428
2014	54	3.759259	4.000000	2.306878	11090	461307
2015	19	3.781955	4.451128	2.375940	102699	460283
2016	8	3.696429	3.625000	2.375000	4976	453237
2017	6	3.071429	3.690476	1.404762	48303	370464

Table 7.11: Rows 1-6 from `disgust_summarised`

7.4 Additional dplyr one-table verbs

Use the code examples below and the help pages to figure out what the following one-table verbs do. Most have pretty self-explanatory names.

7.4.1 rename()

You can rename columns with `rename()`. Set the argument name to the new name, and the value to the old name. You need to put a name in quotes or backticks if it doesn't follow the rules for a good variable name (contains only letter, numbers, underscores, and full stops; and doesn't start with a number).

```
sw <- starwars %>%
  rename(Name = name,
         Height = height,
         Mass = mass,
         `Hair Colour` = hair_color,
         `Skin Colour` = skin_color,
         `Eye Colour` = eye_color,
         `Birth Year` = birth_year)

names(sw)
```

```
##  [1] "Name"        "Height"      "Mass"        "Hair Colour" "Skin
Colour"
##  [6] "Eye Colour"  "Birth Year"  "sex"         "gender"
"homeworld"
## [11] "species"     "films"       "vehicles"    "starships"
```

Almost everyone gets confused at some point with `rename()` and tries to put the original names on the left and the new names on the right. Try it and see what the error message looks like.

7.4.2 distinct()

Get rid of exactly duplicate rows with `distinct()`. This can be helpful if, for example, you are merging data from multiple computers and some of the data got copied from one computer to another, creating duplicate rows.

```
# create a data table with duplicated values
dupes <- tibble(
  id = c( 1,   2,   1,   2,   1,   2),
  dv = c("A", "B", "C", "D", "A", "B")
)

distinct(dupes)
```

id	dv
1	A
2	B
1	C
2	D

7.4.3 count()

The function `count()` is a quick shortcut for the common combination of `group_by()` and `summarise()` used to count the number of rows per group.

```
starwars %>%
  group_by(sex) %>%
  summarise(n = n(), .groups = "drop")
```

sex	n
female	16
hermaphroditic	1
male	60
none	6
NA	4

```
count(starwars, sex)
```

sex	n
female	16
hermaphroditic	1
male	60
none	6
NA	4

7.4.4 slice()

```
slice(starwars, 1:3, 10)
```

name	height	mass	hair_color	skin_color	eye_color	birth_year	sex	genc
Luke Skywalker	172	77	blond	fair	blue	19	male	masc

C-3PO	167	75	NA	gold	yellow	112	none	masc
R2-D2	96	32	NA	white, blue	red	33	none	masc

Obi-Wan Kenobi	182	77	auburn, white	fair	blue-gray	57	male	masc

7.4.5 pull()

```
starwars %>%
  filter(species == "Droid") %>%
  pull(name)
```

```
## [1] "C-3PO"  "R2-D2"  "R5-D4"  "IG-88"  "R4-P17" "BB8"
```

7.5 Window functions

Window functions use the order of rows to calculate values. You can use them to do things that require ranking or ordering, like choose the top scores in each class, or accessing the previous and next rows, like calculating cumulative sums or means.

The dplyr window functions vignette has very good detailed explanations of these functions, but we've described a few of the most useful ones below.

7.5.1 Ranking functions

```
grades <- tibble(
  id = 1:5,
  "Data Skills" = c(16, 17, 17, 19, 20),
  "Statistics"  = c(14, 16, 18, 18, 19)
) %>%
  gather(class, grade, 2:3) %>%
  group_by(class) %>%
  mutate(row_number = row_number(),
         rank       = rank(grade),
         min_rank   = min_rank(grade),
         dense_rank = dense_rank(grade),
         quartile   = ntile(grade, 4),
         percentile = ntile(grade, 100))
```

id	class	grade	row_number	rank	min_rank	dense_rank	quartile	percentil
1	Data Skills	16	1	1.0	1	1	1	
2	Data Skills	17	2	2.5	2	2	1	
3	Data Skills	17	3	2.5	2	2	2	
4	Data Skills	19	4	4.0	4	3	3	
5	Data Skills	20	5	5.0	5	4	4	
1	Statistics	14	1	1.0	1	1	1	
2	Statistics	16	2	2.0	2	2	1	
3	Statistics	18	3	3.5	3	3	2	
4	Statistics	18	4	3.5	3	3	3	
5	Statistics	19	5	5.0	5	4	4	

Table 6.1: All rows from `grades`

- What are the differences among `row_number()`, `rank()`, `min_rank()`, `dense_rank()`, and `ntile()`?
- Why doesn't `row_number()` need an argument?
- What would happen if you gave it the argument `grade` or `class`?
- What do you think would happen if you removed the `group_by(class)` line above?
- What if you added `id` to the grouping?
- What happens if you change the order of the rows?
- What does the second argument in `ntile()` do?

You can use window functions to group your data into quantiles.

```
sw_mass <- starwars %>%
  group_by(tertile = ntile(mass, 3)) %>%
  summarise(min = min(mass),
            max = max(mass),
            mean = mean(mass),
            .groups = "drop")
```

tertile	min	max	mean
1	15	68	45.6600
2	74	82	78.4100
3	83	1358	171.5789
NA	NA	NA	NA

Table 7.12: All rows from `sw_mass`

Why is there a row of `NA` values? How would you get rid of them?

7.5.2 Offset functions

The function `lag()` gives a previous row's value. It defaults to 1 row back, but you can change that with the `n` argument. The function `lead()` gives values ahead of the current row.

```r
lag_lead <- tibble(x = 1:6) %>%
  mutate(lag = lag(x),
         lag2 = lag(x, n = 2),
         lead = lead(x, default = 0))
```

x	lag	lag2	lead
1	NA	NA	2
2	1	NA	3
3	2	1	4
4	3	2	5
5	4	3	6
6	5	4	0

Table 7.13: All rows from `lag_lead`

You can use offset functions to calculate change between trials or where a value changes. Use the `order_by` argument to specify the order of the rows. Alternatively, you can use `arrange()` before the offset functions.

```r
trials <- tibble(
  trial = sample(1:10, 10),
  cond = sample(c("exp", "ctrl"), 10, T),
  score = rpois(10, 4)
) %>%
  mutate(
    score_change = score - lag(score, order_by = trial),
    change_cond = cond != lag(cond, order_by = trial,
                              default = "no condition")
  ) %>%
  arrange(trial)
```

trial	cond	score	score_change	change_cond
1	ctrl	8	NA	TRUE
2	ctrl	4	-4	FALSE
3	exp	6	2	TRUE
4	ctrl	2	-4	TRUE
5	ctrl	3	1	FALSE
6	ctrl	6	3	FALSE

7	ctrl	2	-4	FALSE
8	exp	4	2	TRUE
9	ctrl	4	0	TRUE
10	exp	3	-1	TRUE

Table 7.14: All rows from `trials`

Look at the help pages for `lag()` and `lead()`.

- What happens if you remove the `order_by` argument or change it to `cond`?
- What does the `default` argument do?
- Can you think of circumstances in your own data where you might need to use `lag()` or `lead()`?

7.5.3 Cumulative aggregates

`cumsum()`, `cummin()`, and `cummax()` are base R functions for calculating cumulative means, minimums, and maximums. The dplyr package introduces `cumany()` and `cumall()`, which return `TRUE` if any or all of the previous values meet their criteria.

```r
cumulative <- tibble(
  time = 1:10,
  obs = c(2, 2, 1, 2, 4, 3, 1, 0, 3, 5)
) %>%
  mutate(
    cumsum = cumsum(obs),
    cummin = cummin(obs),
    cummax = cummax(obs),
    cumany = cumany(obs == 3),
    cumall = cumall(obs < 4)
  )
```

time	obs	cumsum	cummin	cummax	cumany	cumall
1	2	2	2	2	FALSE	TRUE
2	2	4	2	2	FALSE	TRUE
3	1	5	1	2	FALSE	TRUE
4	2	7	1	2	FALSE	TRUE
5	4	11	1	4	FALSE	FALSE

Data Skills for Reproducible Research:

DOI 10.5281/zenodo.6527194

Search

Table of contents

6	3	14	1	4	TRUE	FALSE
7	1	15	1	4	TRUE	FALSE
8	0	15	0	4	TRUE	FALSE
9	3	18	0	4	TRUE	FALSE
10	5	23	0	5	TRUE	FALSE

Table 7.15: All rows from `cumulative`

- What would happen if you change `cumany(obs == 3)` to `cumany(obs > 2)`?
- What would happen if you change `cumall(obs < 4)` to `cumall(obs < 2)`?
- Can you think of circumstances in your own data where you might need to use `cumany()` or `cumall()`?

7.6 Glossary

term	definition
data wrangling	The process of preparing data for visualisation and statistical analysis.

7.7 Further Resources

- Chapter 5: Data Transformation in *R for Data Science*
- Data transformation cheat sheet
- Chapter 16: Date and times in *R for Data Science*

« 6 Tidy Data

8 Introduction to GLM »

On this page

"**Data Skills for Reproducible Research**: DOI 10.5281/zenodo.6527194" was written by . It was last built on 2022-05-07.